Report for Programming Problem 1

Team:

Student ID: 2019220082 **Name:** Sofia Santos Neves

Student ID: 2019219581 **Name:** Tatiana Silva Almeida

1. Algorithm description

Começámos por considerar que cada peça são quatro vetores de inteiros, onde cada um desses vetores representa um lado da peça (cima, direita, baixo e esquerda). Colocamos todos os 4 vetores dentro de um único vetor que nomeamos ‘*’sides”*. De modo a sabermos a orientação da peça e sabermos se esta já se encontrava ou não no *“board”*, utilizámos um único inteiro (*‘’is_used”*) que toma valores de 0 a 4. Os valores entre 1 e 4 representam qual dos vetores da peça é que se encontra colocado à esquerda. Já o 0, representa a não utilização da peça.

Começando na função *“main”* o que se fez inicialmente foi ver quantas vezes íamos ter que resolver o problema do “*impossible_puzzle_algorithm”*. De seguida, aplicando um ciclo, lemos cada peça e criamos a estrutura que a representa, inserindo-a em seguida no vetor *‘’pieces”*. Ao mesmo tempo, vamos atualizando o vetor *“count_possibilities”*, responsável por guardar, por cor, o número de vezes que ela aparece ao longo do puzzle.

Enquanto lemos as peças e criamos os vetores pelas quais elas são constituídas, para cada vetor (exemplo: [num1, num2], pertencentes ao vetor *‘’sides”* explicado anteriormente) vamos também atualizar uma matriz *“connects”*, indo à linha num1, coluna num2 e adicionamos um vetor que contém o índice da peça no vetor *‘’pieces”* assim como o número do vetor que estamos a pôr [1-cima, 2-direita, 3-baixo, 4-esquerda]. Com isto, conseguimos sempre saber quais as peças que encaixam em cada vetor.

Em cada iteração do ‘‘*impossible_puzzle_algorithm”* vamos testar quais as peças é que encaixam com as anteriores, já colocadas no tabuleiro previamente. Ao fazer isto, temos em consideração alguns pontos, nomeadamente, se a peça atual:

1. Está na primeira linha - só temos em consideração encontrar uma peça que encaixe à direita da atual;
2. É a última peça - caso base onde só retornamos true;
3. É a última de qualquer linha - temos em consideração qual peça encaixar embaixo da primeira peça da linha atual;
4. É qualquer outra peça do tabuleiro- temos de considerar qual peça encaixa à nossa direita e qual encaixa por baixo da peça da linha anterior na próxima coluna.

Assim que encontramos uma peça que satisfaça os nossos requisitos, vamos colocá-la no tabuleiro e chamar novamente a função ‘‘*impossible_puzzle_algorithm”* para a próxima posição. Contudo, esta pode retornar “*false”* caso não haja nenhuma peça que encaixe nela dada a constituição atual do tabuleiro.Se isso acontecer, a

próxima peça que encaixa vai ser testada e assim sucessivamente, retornando ‘’*false’’* quando todas as possibilidades foram esgotadas sem sucesso.

Relativamente a truques para melhorar a performance do nosso algoritmo decidimos que, antes de colocarmos uma peça no tabuleiro, caso esta não se encontrasse na última linha, era importante vermos se existia alguma peça que encaixasse na parte inferior da mesma. Em caso negativo, poderíamos logo descartar a peça ao invés de a testarmos juntamente com todas as suas sucessões, cortando assim bastantes recursões.

Para além disso, tivemos também em consideração mecanismos que nos permitissem detetar *‘’impossible puzzle!’’* com antecedência durante o pré-processamento. Para isso, analisámos o vetor ‘’*count_possibilities’’* e vimos quantas cores é que tinham uma contagem ímpar. Tendo em conta que as únicas cores que não têm obrigatoriamente um par são aquelas que pertencem aos cantos do tabuleiro, se existissem mais de 4 cores com número ímpar na sua contagem então não era necessário iniciar o ‘’‘*impossible_puzzle_algorithm’’* e devolvemos logo *‘’impossible puzzle!’’.*

É também importante salientar que as escolhas das estruturas de dados foram pensadas de forma a melhorar o tempo de acesso aos valores necessários.

2. Data structures

Existem várias estruturas que contribuíram para a resolução do problema:

- Dois *vector<Piece>* intitulados *pieces* e *board*
  - *pieces* ➞ guarda todas as peças ordenadas por ordem de leitura
  - *board* ➞ guarda todas as peças pela ordem em que são inseridas no tabuleiro
- *vector<int> count_possibilities* ➞ *vetor para guardar, para cada cor, a sua contagem*
- *vector<vector<vector<pair<int, int>>>> connects* ➞ Matriz que vai guardar um vetor com um par de inteiros ([índice_em_pieces, número_vetor]) na linha e coluna do vetor correspondente. Exemplificando:

  Peça1: [1 2 3 4]
  Vetor1: [1,2] **Vetor2:** [2, 3] **Vetor3:** [3,4] **Vetor4:** [4,1]
  Peça2: [2 3 4 1]
  Vetor1: [2,3] **Vetor2:** [3,4] **Vetor3:** [4,1] **Vetor4:** [1,2]

linha/coluna	0	1	2	3	4
0					
1			[1,1], [2,4]		
2				[1,2], [2,1]	
3					[1,3], [2,2]
4		[1,4], [2,3]			

3. Correctness

A nossa abordagem está correta devido aos seguintes fatores:

- Acesso em tempo constante às peças que encaixam num determinado vetor graças ao uso da estrutura ‘’*connects’”*;
- De modo a poupar recursões, (uma vez que construímos o tabuleiro da esquerda para a direita e de cima para baixo) é realizado um corte quando se tenta colocar uma peça no tabuleiro, se a mesma não tiver nenhuma peça que encaixe por baixo ela não será testada;
- Deteção de “impossible puzzle!” em pré-processamento. Como referido anteriormente, quando há mais que quatro cores ímpares destinadas à construção do tabuleiro é porque não vai haver pares suficientes para formar um puzzle. A detetarmos este tipo de casos, imprimimos de imediato ‘’*impossible puzzle!*’’ evitando assim a chamada do algoritmo recursivo.

4. Algorithm Analysis

Complexidade temporal

- do passo recursivo:$O(4(n-1) * 4(n-2) * \ldots * 4(n-n))$
  $\Leftrightarrow O(4^{n-1} * (n-1)!)$

- do caso base: O(1)

Complexidade espacial

- do passo recursivo: $O(2 * (4n) + 1000 * 1000$
  $\Leftrightarrow O(8n)$ -> (assumindo n um número infinito)
  $\Leftrightarrow O(n)$

- do caso base: O(1)

5. References

[1] - https://en.cppreference.com/w/